PUBG Finish Placement Prediction

PlayerUnknown's BattleGrounds (PUBG) has enjoyed massive popularity. With over 50 million copies sold, it's the fifth best selling game of all time, and has millions of active monthly players.

In [ ]:
```
#EE 660 Project
# Created by Binh Phan

# For autoreloading modules
%load_ext autoreload
%autoreload 2
# # For notebook plotting
# %matplotlib inline

#Standard libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os
print(os.listdir("../input"))
# Visualization
import scipy
import matplotlib.pyplot as plt

import seaborn as sns
from scipy.cluster import hierarchy as hc

#Machine Learning

import sklearn
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.decomposition import PCA
# import tensorflow as tf
# from keras.models import Sequential
# from keras.layers import Dense, Dropout, BatchNormalization
# from keras.callbacks import EarlyStopping
# from keras import optimizers
# from keras import regularizers
import xgboost as xgb
# import lightgbm as lgb
```

In [ ]:
```
train = pd.read_csv('../input/train_V2.csv')
```

In [ ]:
```
train.head()
```

In [ ]:
```
# Remove Id, which is not a useful feature
train.drop(columns=['Id'], inplace=True)
```

In [ ]:
```
train.isna().any()
```

In [ ]:
```
train[train['winPlacePerc'].isnull()]
```

In [ ]:
```
train = train.dropna()
train.isna().any()
```

In [ ]:
```
train.info()
```

In [ ]:
```
train.shape
```

In [ ]:
```
print('There are {} different Match types in the dataset.'.format(train['match
Type'].nunique()))

# One hot encode matchType
train = pd.get_dummies(train, columns=['matchType'])

# Take a look at the encoding
matchType_encoding = train.filter(regex='matchType')
matchType_encoding.head()
```

In [ ]:
```
# Turn groupId and match Id into categorical types
train['groupId'] = train['groupId'].astype('category')
train['matchId'] = train['matchId'].astype('category')

# Get category coding for groupId and matchID
train['groupId_cat'] = train['groupId'].cat.codes
train['matchId_cat'] = train['matchId'].cat.codes

# Get rid of old columns
train.drop(columns=['groupId', 'matchId'], inplace=True)

# Lets take a look at our newly created features
train[['groupId_cat', 'matchId_cat']].head()
```

Exploratory Data Analysis

In [ ]:
```
# Take sample for debugging and exploration
sample = 50000
df_sample = train.sample(sample)

# Split sample into training data and target variable
df = df_sample.drop(columns = ['winPlacePerc']) #all columns except target
y = df_sample['winPlacePerc'] # Only target variable
```

In [ ]:
```
# Correlation heatmap
corr = df.corr()

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Create heatmap
heatmap = sns.heatmap(corr)
```

In [ ]:
```
# Create a Dendrogram to view highly correlated features
corr = np.round(scipy.stats.spearmanr(df).correlation, 4)
corr_condensed = hc.distance.squareform(1-corr)
z = hc.linkage(corr_condensed, method='average')
fig = plt.figure(figsize=(14,10))
dendrogram = hc.dendrogram(z, labels=df.columns, orientation='left', leaf_font
_size=16)
plt.plot()
```

Preprocessing

In [ ]:
```
#Import data
train = pd.read_csv('../input/train_V2.csv')

train = train.sample(50000)
X_test = pd.read_csv('../input/test_V2.csv')
# Remove Id, which is not a useful feature
train.drop(columns=['Id'], inplace=True)
train = train.dropna()
# One hot encode matchType
train = pd.get_dummies(train, columns=['matchType'])
X_test = pd.get_dummies(X_test, columns=['matchType'])
# Turn groupId and match Id into categorical types
train['groupId'] = train['groupId'].astype('category')
train['matchId'] = train['matchId'].astype('category')
X_test['groupId'] = X_test['groupId'].astype('category')
X_test['matchId'] = X_test['matchId'].astype('category')

# Get category coding for groupId and matchID
train['groupId_cat'] = train['groupId'].cat.codes
train['matchId_cat'] = train['matchId'].cat.codes
X_test['groupId_cat'] = X_test['groupId'].cat.codes
X_test['matchId_cat'] = X_test['matchId'].cat.codes

# Get rid of old columns
train.drop(columns=['groupId', 'matchId'], inplace=True)
X_test.drop(columns=['groupId', 'matchId'], inplace=True)

# Split train into features and target variable
X_train = train.drop(columns = ['winPlacePerc']) #all columns except target
Y = train['winPlacePerc'] # Only target variable
Y = Y.astype('float32')
x_train, x_val, y_train, y_val = sklearn.model_selection.train_test_split(X_tr
ain, Y, random_state=0, test_size=0.3)

#Standard scaling train features to have 0 mean and 1 variance
columns_to_scale = ['assists', 'boosts', 'damageDealt', 'DBNOs', 'headshotKill
s', 'heals',
       'killPlace', 'killPoints', 'kills', 'killStreaks', 'longestKill',
       'matchDuration', 'maxPlace', 'numGroups', 'rankPoints', 'revives',
       'rideDistance', 'roadKills', 'swimDistance', 'teamKills',
       'vehicleDestroys', 'walkDistance', 'weaponsAcquired', 'winPoints']
categorical  = ['matchType_crashfpp', 'matchType_crashtpp', 'matchType_duo',
       'matchType_duo-fpp', 'matchType_flarefpp', 'matchType_flaretpp',
       'matchType_normal-duo', 'matchType_normal-duo-fpp',
       'matchType_normal-solo', 'matchType_normal-solo-fpp',
       'matchType_normal-squad', 'matchType_normal-squad-fpp',
       'matchType_solo', 'matchType_solo-fpp', 'matchType_squad',
       'matchType_squad-fpp', 'groupId_cat', 'matchId_cat']
train_scale = x_train[columns_to_scale]
train_categorical = x_train[categorical]
val_scale = x_val[columns_to_scale]
val_categorical = x_val[categorical]
test_scale = X_test[columns_to_scale]
test_categorical = X_test[categorical]
# print(df_scale.shape)
# print(df_categorical.shape)
```

```
scaler = sklearn.preprocessing.StandardScaler()
scaler.fit(train_scale.values)

#Standard scaling train features to have 0 mean and 1 variance
train_scale = pd.DataFrame(scaler.transform(train_scale.values), index=train_s
cale.index, columns=train_scale.columns)
# print(df_scale.shape)
x_train = pd.concat([train_scale, train_categorical],axis=1)

#Standard scaling val features with the same parameters in train data
val_scale = pd.DataFrame(scaler.transform(val_scale.values), index=val_scale.i
ndex, columns=val_scale.columns)
# print(df_scale.shape)
x_val = pd.concat([val_scale, val_categorical],axis=1)

#Standard scaling test features with the same parameters in train data
test_scale = pd.DataFrame(scaler.transform(test_scale.values), index=test_scal
e.index, columns=test_scale.columns)
# print(df_scale.shape)

test_id = X_test.loc[:, ['Id']]
X_test = pd.concat([test_scale, test_categorical],axis=1)
```

Reducing Memory

In [ ]:
```
# Thanks and credited to https://www.kaggle.com/gemartin who created this wond
erful mem reducer
def reduce_mem_usage(df):
    """ iterate through all the columns of a dataframe and modify the data typ
e
        to reduce memory usage.
    """
    start_mem = df.memory_usage().sum()
    print('Memory usage of dataframe is {:.2f} MB'.format(start_mem))

    for col in df.columns:
        col_type = df[col].dtype

        if col_type != object:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8)
.max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.in
t16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.in
t32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.in
t64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.fl
oat16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.
float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)
        else:
            df[col] = df[col].astype('category')

    end_mem = df.memory_usage().sum()
    print('Memory usage after optimization is: {:.2f} MB'.format(end_mem))
    print('Decreased by {:.1f}%'.format(100 * (start_mem - end_mem) / start_me
m))

    return df

x_train = reduce_mem_usage(x_train)
x_val = reduce_mem_usage(x_val)
X_test = reduce_mem_usage(X_test)
```

Baseline: Linear Regression

In [ ]:
```
# from sklearn.linear_model import LinearRegression
reg = sklearn.linear_model.LinearRegression().fit(x_train, y_train)
pred_train = reg.predict(x_train)
pred_val = reg.predict(x_val)
mae_train = sklearn.metrics.mean_absolute_error(pred_train, y_train)
mae_val = sklearn.metrics.mean_absolute_error(pred_val, y_val)
print('MAE train: ', mae_train)
print('MAE val: ', mae_val)
```

Boosted Trees Regressor

In [ ]:
```
trees = GradientBoostingRegressor()
```

In [ ]:
```
trees.fit(x_train, y_train)
pred_train = trees.predict(x_train)
pred_val = trees.predict(x_val)
mae_train = sklearn.metrics.mean_absolute_error(pred_train, y_train)
mae_val = sklearn.metrics.mean_absolute_error(pred_val, y_val)
print('MAE train: ', mae_train)
print('MAE val: ', mae_val)
```

In [ ]:
```
rtrees = RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None
,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=3, n_jobs=None,
           oob_score=False, random_state=None, verbose=0, warm_start=False)
rtrees.fit(x_train, y_train)
y_pred = rtrees.predict(x_val)
mae = sklearn.metrics.mean_absolute_error(y_pred, y_val)
mae
```

Keras

In [ ]:
```
# BATCH_SIZE = 256
# EPOCHS = 50
# LEARNING_RATE = 0.001

# model = Sequential()
# model.add(Dense(128, activation='relu', input_dim=x_train.shape[1], activity
_regularizer=regularizers.l1(0.01)))
# # model.add(BatchNormalization())
# # model.add(Dense(128, activation='relu'))
# # model.add(BatchNormalization())
# # model.add(Dense(64, activation='relu'))
# # model.add(BatchNormalization())
# # model.add(Dense(32, activation='relu'))
# # model.add(BatchNormalization())
# model.add(Dense(8, activation='relu'))
# model.add(BatchNormalization())
# model.add(Dense(1))

# adam = optimizers.adam(lr=LEARNING_RATE)
# model.compile(loss='mse', optimizer=adam, metrics=['mae'])
```

In [ ]:
```
# model.summary()
```

In [ ]:
```
# history = model.fit(x=x_train, y=y_train, batch_size=BATCH_SIZE, epochs=20,
#                     verbose=1, validation_data=(x_val,y_val),
#                     shuffle=True)
```

In [ ]:
```
# def plot_loss_accuracy(history):
#     plt.figure(figsize=(10,10))
#     plt.plot(history.history['loss'])
#     plt.plot(history.history['val_loss'])
#     plt.title('model loss')
#     plt.ylabel('loss')
#     plt.xlabel('epoch')
#     plt.legend(['train', 'test'], loc='upper right')
#     plt.show()
# plot_loss_accuracy(history)
```

In [ ]:
```
# X_test.shape
```

In [ ]:
```
# prediction = model.predict(X_test, batch_size=128, verbose=1)
```

In [ ]:
```
# prediction = prediction.flatten()
```

In [ ]:
```
# prediction.shape
```

In [ ]:
```
# pred_df = pd.DataFrame({'Id' : test_id['Id'], 'winPlacePerc' : prediction})

# # Create submission file
# pred_df.to_csv("submission.csv", index=False)
```

Deep Neural Network

In [ ]:
```
# # Model parameters
# BATCH_SIZE = 512
# STEPS = 40000
# LEARNING_RATE = 0.001
# HIDDEN_UNITS = [256, 128, 64, 32]
```

In [ ]:
```
# feature_columns = [tf.feature_column.numeric_column(x) for x in x_train.colu
mns]

# train_fn = tf.estimator.inputs.pandas_input_fn(x_train, y_train, shuffle = T
rue)
# val_fn = tf.estimator.inputs.pandas_input_fn(x_val, y_val, shuffle = False)
# test_fn = tf.estimator.inputs.pandas_input_fn(X_test, None, shuffle = False)
```

In [ ]:
```
# # optimizer = tf.train.AdamOptimizer(learning_rate=LEARNING_RATE)
# optimizer = tf.train.ProximalAdagradOptimizer(learning_rate=0.1, l1_regulari
zation_strength=0.001, l2_regularization_strength=0.001)
# estimator = tf.estimator.DNNLinearCombinedRegressor(
#     dnn_feature_columns=feature_columns,
#     dnn_hidden_units=HIDDEN_UNITS,
#     dnn_optimizer=optimizer)
# train_spec = tf.estimator.TrainSpec(train_fn, max_steps=STEPS)
# eval_spec = tf.estimator.EvalSpec(val_fn, steps=500, throttle_secs=300)
```

In [ ]:
```
# tf.estimator.train_and_evaluate(estimator, train_spec=train_spec, eval_spec=
eval_spec)
```

In [ ]:
```
# prediction = estimator.predict(test_fn)
```

In [ ]:
```
# prediction_df = pd.DataFrame(prediction)
```

In [ ]:
```
# prediction_df.values.flatten()
```

In [ ]:
```
# pred_df = pd.DataFrame({'Id' : test_id['Id'], 'winPlacePerc' : prediction_d
f.values.flatten()})

# # Create submission file
# pred_df.to_csv("submission.csv", index=False)
```

In [ ]:
```
# def mae(labels, predictions):
#     pred_values = predictions['predictions']
# #     print type(pred_values)
#     return {'mae': tf.metrics.mean_absolute_error(labels, pred_values)}
```

In [ ]:
```
# DNN = tf.estimator.DNNRegressor(
#     feature_columns=feature_columns,
#     hidden_units=[2048, 1024,256, 128, 64, 32,4,1],
#     dropout = 0.1,
#     optimizer=tf.train.ProximalAdagradOptimizer(
#       learning_rate=0.1,
#       l1_regularization_strength=0.001
#     ))
# DNN = tf.contrib.estimator.add_metrics(DNN, mae)
```

In [ ]:
```
# DNN.train(train_fn)
```

In [ ]:
```
# DNN.evaluate(val_fn)
```

In [ ]:
```
# predictions = np.array([item['predictions'][0] for item in preds]).astype(n
p.float64)
```

In [ ]:
```
# yvalnp = np.array(y_val)
```

In [ ]:
```
# type(yvalnp[0])
```

In [ ]:
```
# type(predictions)
```

In [ ]:
```
# mae = sklearn.metrics.mean_absolute_error(yvalnp, predictions)
```

In [ ]:
```
# mae
```

XGBoost

Preprocessing

In [ ]:
```
#Import data

train = pd.read_csv('../input/train_V2.csv')

train = train.sample(100000)
X_test = pd.read_csv('../input/test_V2.csv')
# Remove Id, which is not a useful feature
train.drop(columns=['Id'], inplace=True)
train = train.dropna()
# One hot encode matchType
train = pd.get_dummies(train, columns=['matchType'])
X_test = pd.get_dummies(X_test, columns=['matchType'])
# Turn groupId and match Id into categorical types
train['groupId'] = train['groupId'].astype('category')
train['matchId'] = train['matchId'].astype('category')
X_test['groupId'] = X_test['groupId'].astype('category')
X_test['matchId'] = X_test['matchId'].astype('category')

# Get category coding for groupId and matchID
train['groupId_cat'] = train['groupId'].cat.codes
train['matchId_cat'] = train['matchId'].cat.codes
X_test['groupId_cat'] = X_test['groupId'].cat.codes
X_test['matchId_cat'] = X_test['matchId'].cat.codes

# Get rid of old columns
train.drop(columns=['groupId', 'matchId'], inplace=True)
X_test.drop(columns=['groupId', 'matchId'], inplace=True)

# Split train into features and target variable
X_train = train.drop(columns = ['winPlacePerc']) #all columns except target
Y = train['winPlacePerc'] # Only target variable
Y = Y.astype('float32')

#Standard scaling train features to have 0 mean and 1 variance
columns_to_scale = ['assists', 'boosts', 'damageDealt', 'DBNOs', 'headshotKill
s', 'heals',
       'killPlace', 'killPoints', 'kills', 'killStreaks', 'longestKill',
       'matchDuration', 'maxPlace', 'numGroups', 'rankPoints', 'revives',
       'rideDistance', 'roadKills', 'swimDistance', 'teamKills',
       'vehicleDestroys', 'walkDistance', 'weaponsAcquired', 'winPoints']
categorical  = ['matchType_crashfpp', 'matchType_crashtpp', 'matchType_duo',
       'matchType_duo-fpp', 'matchType_flarefpp', 'matchType_flaretpp',
       'matchType_normal-duo', 'matchType_normal-duo-fpp',
       'matchType_normal-solo', 'matchType_normal-solo-fpp',
       'matchType_normal-squad', 'matchType_normal-squad-fpp',
       'matchType_solo', 'matchType_solo-fpp', 'matchType_squad',
       'matchType_squad-fpp', 'groupId_cat', 'matchId_cat']
train_scale = X_train[columns_to_scale]
train_categorical = X_train[categorical]
test_scale = X_test[columns_to_scale]
test_categorical = X_test[categorical]
# print(df_scale.shape)
# print(df_categorical.shape)
scaler = sklearn.preprocessing.StandardScaler()
scaler.fit(train_scale.values)
```

```
#Standard scaling train features to have 0 mean and 1 variance
train_scale = pd.DataFrame(scaler.transform(train_scale.values), index=train_s
cale.index, columns=train_scale.columns)
# print(df_scale.shape)
X_train = pd.concat([train_scale, train_categorical],axis=1)

#Standard scaling test features with the same parameters in train data
test_scale = pd.DataFrame(scaler.transform(test_scale.values), index=test_scal
e.index, columns=test_scale.columns)
# print(df_scale.shape)

test_id = X_test.loc[:, ['Id']]
X_test = pd.concat([test_scale, test_categorical],axis=1)
```

Reducing Memory

In [ ]:
```
# Thanks and credited to https://www.kaggle.com/gemartin who created this wond
erful mem reducer
def reduce_mem_usage(df):
    """ iterate through all the columns of a dataframe and modify the data typ
e
        to reduce memory usage.
    """
    start_mem = df.memory_usage().sum()
    print('Memory usage of dataframe is {:.2f} MB'.format(start_mem))

    for col in df.columns:
        col_type = df[col].dtype

        if col_type != object:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8)
.max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.in
t16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.in
t32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.in
t64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.fl
oat16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.
float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)
        else:
            df[col] = df[col].astype('category')

    end_mem = df.memory_usage().sum()
    print('Memory usage after optimization is: {:.2f} MB'.format(end_mem))
    print('Decreased by {:.1f}%'.format(100 * (start_mem - end_mem) / start_me
m))

    return df

X_train = reduce_mem_usage(X_train)
X_test = reduce_mem_usage(X_test)
```

Cross-Validation

In [ ]:
```
#Cross-validation
params = {
    # Parameters that we are going to tune.
    'eta': 0.28, #Result of tuning with CV
    'max_depth': 5, #Result of tuning with CV
    'subsample': 1, #Result of tuning with CV
    'lambda': 0.01, #Result of tuning with CV
    'colsample_bytree': 0.5, #Result of tuning with CV
    # Other parameters
    'objective':'reg:linear',
    'eval_metric':'mae',
    'silent': 1
}


#Block of code used for hypertuning parameters. Adapt to each round of paramet
er tuning.
#Turn off CV in submission
CV=False
if CV:
    dtrain = xgb.DMatrix(X_train,label=Y)
    gridsearch_params = {
        'eta': [(eta) for eta in np.arange(.04, 0.3, .02)],
        'max_depth': [(max_depth) for max_depth in np.arange(1,6,1)],
        'min_child_weight': [(min_child_weight) for min_child_weight in np.ara
nge(1,6,1)],
        'subsample': [(subsample) for subsample in np.arange(0,1.1,0.5)],
        'lambda': [(lambd) for lambd in np.geomspace(0.01,10,num=5)],
        'alpha': [(alpha) for alpha in np.geomspace(0.01,10,num=5)]
    }

    # Define initial best params and MAE
    min_mae = float("Inf")
    best_params = {}

    #searching for best eta
    for param in gridsearch_params:
        print("********************************CV with param {} **************
**************************".format(
                                    param))
        for i in gridsearch_params[param]:
            print('now at ', i)
            # Update our parameters
            params[param] = i

            # Run CV
            cv_results = xgb.cv(
                params,
                dtrain,
                num_boost_round=100,
                nfold=3,
                metrics={'mae'},
                early_stopping_rounds=10
            )
#             for result in cv_results:
#                 print(cv_results[result])
```

```
            # Update best MAE
            mean_mae = cv_results['test-mae-mean'].min()
            boost_rounds = cv_results['test-mae-mean'].argmin()
            print("MAE {} for {} rounds".format(mean_mae, boost_rounds+1))
            if mean_mae < min_mae:
                min_mae = mean_mae
                print(set(gridsearch_params).intersection(set(params)))
                for k in set(gridsearch_params).intersection(set(params)):
                    best_params[k] = params[k]
            params.pop(param, None)
        if param in best_params:
            params[param] = best_params[param]

    print("Best params: {}, MAE: {}".format(best_params, min_mae))
else:
    #Print final params to use for the model
    params['silent'] = 0 #Turn on output
    print(params)
```

In [ ]:
```
params
```

Training on all Data using params

In [ ]:
```
#Import data
train = pd.read_csv('../input/train_V2.csv')
X_test = pd.read_csv('../input/test_V2.csv')
# Remove Id, which is not a useful feature
train.drop(columns=['Id'], inplace=True)
train = train.dropna()
# One hot encode matchType
train = pd.get_dummies(train, columns=['matchType'])
X_test = pd.get_dummies(X_test, columns=['matchType'])
# Turn groupId and match Id into categorical types
train['groupId'] = train['groupId'].astype('category')
train['matchId'] = train['matchId'].astype('category')
X_test['groupId'] = X_test['groupId'].astype('category')
X_test['matchId'] = X_test['matchId'].astype('category')

# Get category coding for groupId and matchID
train['groupId_cat'] = train['groupId'].cat.codes
train['matchId_cat'] = train['matchId'].cat.codes
X_test['groupId_cat'] = X_test['groupId'].cat.codes
X_test['matchId_cat'] = X_test['matchId'].cat.codes

# Get rid of old columns
train.drop(columns=['groupId', 'matchId'], inplace=True)
X_test.drop(columns=['groupId', 'matchId'], inplace=True)

# Split train into features and target variable
X_train = train.drop(columns = ['winPlacePerc']) #all columns except target
Y = train['winPlacePerc'] # Only target variable
Y = Y.astype('float32')

#Standard scaling train features to have 0 mean and 1 variance
columns_to_scale = ['assists', 'boosts', 'damageDealt', 'DBNOs', 'headshotKill
s', 'heals',
       'killPlace', 'killPoints', 'kills', 'killStreaks', 'longestKill',
       'matchDuration', 'maxPlace', 'numGroups', 'rankPoints', 'revives',
       'rideDistance', 'roadKills', 'swimDistance', 'teamKills',
       'vehicleDestroys', 'walkDistance', 'weaponsAcquired', 'winPoints']
categorical  = ['matchType_crashfpp', 'matchType_crashtpp', 'matchType_duo',
       'matchType_duo-fpp', 'matchType_flarefpp', 'matchType_flaretpp',
       'matchType_normal-duo', 'matchType_normal-duo-fpp',
       'matchType_normal-solo', 'matchType_normal-solo-fpp',
       'matchType_normal-squad', 'matchType_normal-squad-fpp',
       'matchType_solo', 'matchType_solo-fpp', 'matchType_squad',
       'matchType_squad-fpp', 'groupId_cat', 'matchId_cat']
train_scale = X_train[columns_to_scale]
train_categorical = X_train[categorical]
test_scale = X_test[columns_to_scale]
test_categorical = X_test[categorical]
# print(df_scale.shape)
# print(df_categorical.shape)
scaler = sklearn.preprocessing.StandardScaler()
scaler.fit(train_scale.values)

#Standard scaling train features to have 0 mean and 1 variance
train_scale = pd.DataFrame(scaler.transform(train_scale.values), index=train_s
cale.index, columns=train_scale.columns)
```

```
# print(df_scale.shape)
X_train = pd.concat([train_scale, train_categorical],axis=1)

#Standard scaling test features with the same parameters in train data
test_scale = pd.DataFrame(scaler.transform(test_scale.values), index=test_scal
e.index, columns=test_scale.columns)
# print(df_scale.shape)

test_id = X_test.loc[:, ['Id']]
X_test = pd.concat([test_scale, test_categorical],axis=1)

X_train = reduce_mem_usage(X_train)
X_test = reduce_mem_usage(X_test)
```

In [ ]:
```
matrix_train = xgb.DMatrix(X_train,label=Y)
model=xgb.train(params=params,
                dtrain=matrix_train,num_boost_round=100)
```

In [ ]:
```
prediction = model.predict(xgb.DMatrix(X_test), ntree_limit = model.best_ntree
_limit)
```

In [ ]:
```
prediction = prediction.flatten()
```

In [ ]:
```
df_sub = pd.read_csv("../input/sample_submission_V2.csv")
df_test = pd.read_csv("../input/test_V2.csv")
df_sub['winPlacePerc'] = prediction
# Restore some columns
df_sub = df_sub.merge(df_test[["Id", "matchId", "groupId", "maxPlace", "numGro
ups"]], on="Id", how="left")

# Sort, rank, and assign adjusted ratio
df_sub_group = df_sub.groupby(["matchId", "groupId"]).first().reset_index()
df_sub_group["rank"] = df_sub_group.groupby(["matchId"])["winPlacePerc"].rank
()
df_sub_group = df_sub_group.merge(
    df_sub_group.groupby("matchId")["rank"].max().to_frame("max_rank").reset_i
ndex(),
    on="matchId", how="left")
df_sub_group["adjusted_perc"] = (df_sub_group["rank"] - 1) / (df_sub_group["nu
mGroups"] - 1)

df_sub = df_sub.merge(df_sub_group[["adjusted_perc", "matchId", "groupId"]], o
n=["matchId", "groupId"], how="left")
df_sub["winPlacePerc"] = df_sub["adjusted_perc"]

# Deal with edge cases
df_sub.loc[df_sub.maxPlace == 0, "winPlacePerc"] = 0
df_sub.loc[df_sub.maxPlace == 1, "winPlacePerc"] = 1

# Align with maxPlace
# Credit: https://www.kaggle.com/anycode/simple-nn-baseline-4
subset = df_sub.loc[df_sub.maxPlace > 1]
gap = 1.0 / (subset.maxPlace.values - 1)
new_perc = np.around(subset.winPlacePerc.values / gap) * gap
df_sub.loc[df_sub.maxPlace > 1, "winPlacePerc"] = new_perc

# Edge case
df_sub.loc[(df_sub.maxPlace > 1) & (df_sub.numGroups == 1), "winPlacePerc"] =
0
assert df_sub["winPlacePerc"].isnull().sum() == 0

df_sub[["Id", "winPlacePerc"]].to_csv("submission_adjusted.csv", index=False)
```

In [ ]:
```
# imp = pd.DataFrame(list(model.get_fscore().items()), columns=['cols', 'im
p'])
# imp['imp'] = imp['imp'] / imp['imp'].sum()
# imp = imp.sort_values('imp', ascending=False)
# for i in range(imp.shape[0]):
#     print(imp['imp'][:i+1].sum())
#     if imp['imp'][:i+1].sum() > 0.9:
#         max_features = i
#         break
# max_features
# imp = imp[:max_features]
# imp
```

In [ ]:
```
# x_traink = x_train[imp['cols']]
# x_valk = x_val[imp['cols']]
```

In [ ]:
```
# model = XGBmodel(x_traink,x_valk,y_train,y_val,params)
```

In [ ]:
```
# trees = GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=
None,
#              learning_rate=1, loss='ls', max_depth=3, max_features=None,
#              max_leaf_nodes=None, min_impurity_decrease=0.0,
#              min_impurity_split=None, min_samples_leaf=1,
#              min_samples_split=2, min_weight_fraction_leaf=0.0,
#              n_estimators=1, presort='auto', random_state=None,
#              subsample=1.0, verbose=0, warm_start=False)
# trees.fit(x_traink, y_train)
# y_pred = trees.predict(x_valk)
# mae = sklearn.metrics.mean_absolute_error(y_pred, y_val)
# mae
```

In [ ]:
```
# m2 = RandomForestRegressor(n_estimators=80, min_samples_leaf=3, max_features
='sqrt',
#                            n_jobs=-1)
# m2.fit(x_traink, y_train)
# y_pred = m2.predict(x_valk)
# mae = sklearn.metrics.mean_absolute_error(y_pred, y_val)
# mae
```

LightGBM

In [ ]:
```
# params = {"objective" : "regression", "metric" : "mae", 'n_estimators':2000
0, 'early_stopping_rounds':200,
#               "num_leaves" : 31, "learning_rate" : 0.05, "bagging_fraction"
 : 0.7,
#                "bagging_seed" : 0, "num_threads" : 4,"colsample_bytree" : 0.
7
#              }

# lgtrain = lgb.Dataset(x_train, label=y_train)
# lgval = lgb.Dataset(x_val, label=y_val)
# model = lgb.train(params, lgtrain, valid_sets=[lgtrain, lgval], early_stoppi
ng_rounds=200, verbose_eval=1000)
```

In [ ]:
```
# params = {"objective" : "regression", "metric" : "mae", 'n_estimators':2000
0, 'early_stopping_rounds':200,
#               "num_leaves" : 31, "learning_rate" : 0.05, "bagging_fraction"
 : 0.7,
#                "bagging_seed" : 0, "num_threads" : 4,"colsample_bytree" : 0.
7
#             }

# lgtrain = lgb.Dataset(x_traink, label=y_train)
# lgval = lgb.Dataset(x_valk, label=y_val)
# model = lgb.train(params, lgtrain, valid_sets=[lgtrain, lgval], early_stoppi
ng_rounds=200, verbose_eval=1000)
```

In [ ]:
```
# pred_test_y = model.predict(X_test, num_iteration=model.best_iteration)
```

In [ ]:
```
# df_sub = pd.read_csv("../input/sample_submission_V2.csv")
# df_test = pd.read_csv("../input/test_V2.csv")
# df_sub['winPlacePerc'] = pred_test_y
# # Restore some columns
# df_sub = df_sub.merge(df_test[["Id", "matchId", "groupId", "maxPlace", "numG
roups"]], on="Id", how="left")

# # Sort, rank, and assign adjusted ratio
# df_sub_group = df_sub.groupby(["matchId", "groupId"]).first().reset_index()
# df_sub_group["rank"] = df_sub_group.groupby(["matchId"])["winPlacePerc"].ran
k()
# df_sub_group = df_sub_group.merge(
#     df_sub_group.groupby("matchId")["rank"].max().to_frame("max_rank").reset
_index(),
#     on="matchId", how="left")
# df_sub_group["adjusted_perc"] = (df_sub_group["rank"] - 1) / (df_sub_group
["numGroups"] - 1)

# df_sub = df_sub.merge(df_sub_group[["adjusted_perc", "matchId", "groupId"]],
 on=["matchId", "groupId"], how="left")
# df_sub["winPlacePerc"] = df_sub["adjusted_perc"]

# # Deal with edge cases
# df_sub.loc[df_sub.maxPlace == 0, "winPlacePerc"] = 0
# df_sub.loc[df_sub.maxPlace == 1, "winPlacePerc"] = 1

# # Align with maxPlace
# # Credit: https://www.kaggle.com/anycode/simple-nn-baseline-4
# subset = df_sub.loc[df_sub.maxPlace > 1]
# gap = 1.0 / (subset.maxPlace.values - 1)
# new_perc = np.around(subset.winPlacePerc.values / gap) * gap
# df_sub.loc[df_sub.maxPlace > 1, "winPlacePerc"] = new_perc

# # Edge case
# df_sub.loc[(df_sub.maxPlace > 1) & (df_sub.numGroups == 1), "winPlacePerc"]
 = 0
# assert df_sub["winPlacePerc"].isnull().sum() == 0

# df_sub[["Id", "winPlacePerc"]].to_csv("submission_adjusted.csv", index=Fals
e)
```